METODOLOGIA ORIENTADA AO PROJETISTA PARA SYSTEM-LEVEL DESIGN

Reinaldo Silveira e Wilhelmus A. M. Van Noije

LSI – Depart. de Engenharia de Sistemas Eletrônicos da
Escola Politécnica da Universidade de São Paulo,
Av. Prof. Luciano Gualberto n.158, trav. 3, Cidade Universitária,
05508-900, São Paulo, SP, Brasil.
(silveira, noije)@lsi.usp.br

ABSTRACT

A great deal of the discussion about design of new EDA tools and methodologies concern about the conciliation between designers needs and programmers capabilities. This work addresses this problem, proposing a new design methodology that instructs how to integrate different tools into a common framework and which are the requirements of this framework to assure productivity at the system level design. This approach is called "Designer Oriented Methodology" (DO), and uses the so called "Game Like Development" concept (GLD) and the pure object-oriented language **Self** as the main element for this integration.

RESUMO

Uma grande preocupação sobre o desenvolvimento de ferramentas de auxílio a projeto eletrônico (EDA), atualmente, refere-se a conciliação entre as necessidades dos projetistas e os interesses dos programadores. Este trabalho aborda este problema, propondo uma nova metodologia de projeto que sugere como pode ser a integração de diferentes ferramentas num *framework* comum e quais devem ser os requisitos deste *framework* para que seja assegurada a produtividade no nível de projeto de sistema. Esta proposta chama-se "*Designer Oriented Methodology*" (DO), e emprega uma série de conceitos e abstrações inclusive a que chamamos de "*Game Like Development*" (GLD). Como elemento principal para a implementação da proposta usa-se a linguagem orientada a objeto **Self**.

METODOLOGIA ORIENTADA AO PROJETISTA PARA SYSTEM-LEVEL DESIGN

Reinaldo Silveira e Wilhelmus A. M. Van Noije

LSI – Departamento de Engenharia de Sistemas Eletrônicos da
Escola Politécnica da Universidade de São Paulo,
Av. Prof. Luciano Gualberto, n.158, trav. 3, Cidade Universitária,
05508-900, São Paulo, SP, Brasil.
(silveira, noije)@lsi.usp.br

RESUMO

Uma grande preocupação sobre o desenvolvimento de ferramentas de auxílio a projeto eletrônico (EDA), atualmente, refere-se a conciliação entre as necessidades dos projetistas e os interesses dos programadores. Este trabalho aborda este problema, propondo uma nova metodologia de projeto que sugere como pode ser a integração de diferentes ferramentas num *framework* comum e quais devem ser os requisitos deste *framework* para que seja assegurada a produtividade no nível de projeto de sistema. Esta proposta chama-se "*Designer Oriented Methodology*" (DO), e emprega uma série de conceitos e abstrações inclusive a que chamamos de "*Game Like Development*" (GLD). Como elemento principal para a implementação da proposta usa-se a linguagem orientada a objeto **Self**.

1. INTRODUÇÃO

Atualmente, o estado-da-arte em ferramentas de projeto digital encontra-se numa situação um tanto difícil devido ao rápido desenvolvimento das tecnologias de fabricação de circuitos integrados e da pressão do Mercado por CI's e sistemas mais complexos, desempenhos cada vez maiores num tempo muito menor. Acompanhando a bibliografia do assunto, notamos a proliferação de impressionantes frases de efeito como "*Systems on a Chip*" (SoCs), "*Intellectual Properties*" (IPs), "*High Level Synthesis*" (HLS), e somos levados a acreditar que o problema de desenvolvimento destes novos sistemas está bem entendido e resolvido. Entretanto, constatamos que isto é conseguido a custos bastante elevados[5].

As metodologias em uso atualmente são basicamente as mesmas propostas de quinze a vinte anos atrás, muitas adaptações foram introduzidas para acomodar desenvolvimentos mais recentes, entretanto pouco ou nenhum aperfeiçoamento real foi introduzido. Estas metodologias são muito difíceis e caras de serem mantidas[3], não é raro uma *design house* manter até cerca de 10% do pessoal somente para a manutenção do fluxo de projeto. A fim de contornar estes problemas é necessário o rompimento com as atuais metodologias e introduzir uma nova, que seja capaz de lidar com todos os problemas atuais de forma elegante e oferecer uma nova perspectiva de evolução a medida que novos métodos e algoritmos forem surgindo. Este trabalho propõe o que acreditamos ser esta nova metodologia. Chamada de "Metodologia Orientada ao Projetista" (DO, *Designer Oriented methodology*), ela consiste num ambiente capaz de oferecer ao usuário/projetista condições apropriadas para que o mesmo possa trabalhar livre de interrupções e distrações, mantendo-o sempre atento e em prontidão para resolver qualquer problema que venha a ocorrer assim que o mesmo seja percebido. Nesta metodologia, todas as informações são apresentadas de forma a reforçar a atenção do usuário em aspectos específicos do projeto. Este reforço deve ser implementado em todas as formas possíveis, assim como um jogo que captura as atenções dos jogadores durante o seu curso.

Este trabalho não se trata de um novo algoritmo de simulação, síntese ou verificação, mas uma nova proposta de integração de todas estas ferramentas e possivelmente muitas outras num *framework* que possa evoluir ao longo do tempo de acordo com as necessidades do projeto. Este *framework* é baseado no que achamos ser mais adequado aos projetistas mantendo-os focados no processo de desenvolvimento.

As próximas seções são divididas da seguinte forma: a seção 2, apresentará uma visão geral das tendências atuais de desenvolvimento de ferramentas de auxílio ao projeto eletrônico (EDA). Estas tendências estão bem estabelecidas por vários anos e representam muito bem a forma com que os projetos são feitos nos dias de hoje. Na seção 3, apresentaremos as razões que nos levaram a elaboração da metodologia orientada ao projetista, e quais os seus requisitos necessários para a sua implementação. Na seção 4, introduziremos o conceito de Jogo e como este pode ser aproveitado na perspectiva do Projeto. Na

seção 5, mostraremos como usando uma linguagem de alto nível podemos implementar um *framework* que adere às especificações da metodologia. Finalmente na seção 6, concluiremos este artigo com as considerações finais sobre o que foi abordado.

2. CONSIDERAÇÕES SOBRE AS FERRAMENTAS DE PROJETO ATUAIS

Desde o início do projeto de computadores, as linguagens de programação tem sido usadas na modelagem de sistemas digitais complexos [8]. Um grande número de técnicas foram elaboradas para modelar os primeiros sistemas, mas cada novo desenvolvimento parecia sempre uma nova experiência, principalmente pela dificuldade de uso destas linguagens numa tarefa tão especifica. Havia um consenso que somente uma linguagem especialmente desenhada para a descrição de *hardware* poderia ser de real utilidade para a industria. Somente em meados da década de 80 surgiram os primeiros padrões industriais de linguagens de descrição de *hardware* (HDL), com o VHDL e o Verilog. Logo, estes padrões tornaram-se comuns nos principais fluxos de projeto. Ao contrário da captura esquemática, a descrição textual é muito mais rápida de ser editada, inequívoca e capaz de descrever o sistema em diferentes níveis de abstração. Isto fez com que fossem estabelecidos níveis de representação apropriados para procedimentos de síntese lógica (*register transfer level* - RTL), e níveis de abstração mais elevados para representação e síntese. Seguindo as HDLs, as ferramentas de síntese foram o próximo grande adendo aos fluxos de projeto. Primeiro as ferramentas de síntese lógica, depois as de síntese de alto nível, apesar desta última ainda não gozar da mesma popularidade da primeira.

Apesar de seu grande sucesso no meio industrial, as HDLs padrão não tiveram sucesso em preencher todas as expectativas especialmente da comunidade acadêmica. Suas dificuldades em lidar com diferentes estruturas/tipos de dados e as limitações sintáticas em descrever abstrações de níveis mais elevados sinalizavam a necessidade para novos tipos de descrição. Muitos grupos afirmam que o caminho é o uso de linguagens de programação de propósito geral, assim como era feito no início, com uma certa ênfase nas linguagens mais populares, como C/C++ [9,10] ou Java [6]. A idéia é usar o poder e flexibilidade destas linguagens para desenvolver um subconjunto (funções/objetos) apropriado para as descrições de *hardware*. Padronizando estes subconjuntos, seria possível utilizar a mesma estrutura de desenvolvimento de *software* para o projeto de sistemas. Outro ponto importante, tendo sido padronizado estes subconjunto de funções, o reuso e a troca de IPs estaria garantida. Além disso, por se tratar de uma linguagem de uso geral a integração em diferentes fluxos de projeto estaria praticamente garantida. Usando a orientação a objetos para esconder a complexidade dos modelos de *hardware* dos programadores/projetistas seria possível evitar os problemas apresentados no início desta prática. A complexidade do projeto digital tem aumentado consideravelmente nos últimos anos, novas classes de problemas aparecem a cada dia. Sem falar no *time-to-market* cada vez menor e dos custos envolvidos. O Mercado apresenta-se muito mais agressivo devido a um novo perfil de consumidor acostumado a inovações tecnológicas a cada ano [5]. A preocupação da industria de semicondutores tem se concentrado nos problemas decorrentes da complexidade ao nível de sistemas, como: reuso, verificação e teste, gerenciamento de projeto, *software* embarcado, otimizações baseadas em custo, e etc. Seguindo estas tendências, os esforços dos meios de pesquisa tem sido em aumentar a produtividade das ferramentas de auxílio (EDA) e fluxos de projetos, introduzindo modelos de descrição de alto nível para sistemas digitais e novos algoritmos de síntese.

O resultado deste processo é uma grande diversidade de ferramentas, modelos e interesses conflitantes que é muito difícil de manter gerenciável e atualizado. Outro problema serio é que estas ferramentas foram desenvolvidas com técnicas da ciência da computação, visando um aspecto muito particular do problema e quase sempre sem considerar as formas de utilização das mesmas. Estas condições, na maioria das vezes, não condiz com a realidade das equipes de desenvolvimento. O ponto de vista do projetista tem sido sistematicamente desconsiderado quando são propostas novas metodologias. Este estado de coisas pode não ser intencional, mas tem aumentado os requisitos das equipes de desenvolvimento que precisam se capacitar em conhecimentos/conceitos de programação algumas vezes muito além do escopo das suas especializações. Isto acaba por aumentar ainda mais o custo com pessoal.

Note que isto não é uma simples crítica a pesquisa da ciência da computação em ferramentas EDA, na realidade eles tem feito uma notável contribuição a todo o sistema de projeto. A questão é que a pesquisa deve prestar mais atenção a forma de interação das ferramentas com os seus usuários, seus problemas e limitações, e da forma com que as estruturas de dados são processadas a apresentadas no fluxo de projeto. A tripla de operações "edição-compilação-simulação" não é indicada quando se trabalha com altos níveis de abstração. Nestes níveis, as mudanças devem ser implementadas e verificadas rapidamente, de forma a não desviar a atenção do projetista do objeto de seu trabalho. Apesar de todos os avanços, a imaginação do usuário ainda desempenha um fator determinante no processo de desenvolvimento. As ferramentas devem

refletir esta importância, assim como todo o fluxo de projeto.

Com todos estes pontos em mente, este trabalho propõe diretrizes para o que pode vir a ser uma nova metodologia de projeto de sistemas digitais. Uma metodologia que adote o ponto de vista do projetista, onde cada ferramenta assuma o agente humano como fonte/guia principal de projeto. Isto pode ser conseguido aplicando o conceito de Jogo (GLD) no desenvolvimento de um *framework* comum que integre as mais avançadas ferramentas, assim como as mais tradicionais, mas que acima de tudo seja diferente na forma com que tais ferramentas interagem com o agente humano ao longo do processo de desenvolvimento.

3. Desenvolvimento Orientado ao Projetista

Na seção anterior afirmamos que as ferramentas de auxílio ao projeto são desenvolvidas sob uma visão muito particular de seus criadores. Isto pode ser constatado facilmente quando analisados o modo operacional das mesmas. Tomemos por exemplo um simulador hipotético. Existem muitos tipos de simulação, a que estamos interessados são os que se referem às simulações arquiteturais e RTL. Nestes casos, cada nível hierárquico corresponde um conjunto de primitivas que constituem a base do processo de simulação. Os simuladores também podem ser do tipo *Levelized Compiled Code* (LCC) ou interpretados [7]; ou quanto ao tipo de inferência de simulação podem ser *event-driven* ou *cycle based*. Em todos estes casos, eles compartilham um ponto em comum: a visão (cultural) dos seus criadores. Vejamos a forma como utilizamos estes simuladores: independente do tipo, *event-driven* ou *cycle based*, os simuladores compilados partem de uma descrição do circuito em termos de primitivas, que é compilada gerando um programa que corresponde ao simulador para aquele circuito. Este programa é então executado em função de entradas que correspondem às condições em que o circuito deve ser verificado. Podemos ver aqui um padrão de operação, que devemos concordar, faz um perfeito paralelo com a forma de desenvolver programas, guardada as devidas proporções.

Este padrão é o que aparece em todos os ciclos de desenvolvimento de programas, ou seja "entrada-filtro-resultado". "Entradas e resultados" correspondem a arquivos com funções especiais, "filtro" corresponde a um agente de transformação que operando sobre o arquivo de entrada produz uma seqüência de dados de saída que é armazenado em "resultados". Isto pode ser observado em quase todo o universo da computação contemporânea, sendo um dos fundamentos dos sistemas UNIX. Na compilação, "entrada" corresponde ao arquivo fonte, o compilador ao "filtro" e o programa executável ao "resultado". No exemplo do simulador compilado observamos a mesma coisa: "descrição-compilação-simulador", respectivamente. Se tomarmos como exemplo um simulador interpretado veremos o mesmo padrão: "descrição-simulador-saída". E assim sucessivamente, quando analisamos outras ferramentas de auxílio. Podemos identificar em cada caso o padrão acima, demonstrando que por mais criativos que os programadores procuram ser, eles ainda estão contaminados com um "modus operandi" surgido a mais de 30 anos e que não corresponde com a realidade dos usuários de seus programas.

A cerca de 15 anos, deu-se início uma difusão mais intensa das interfaces gráficas e "amigáveis" nos sistemas de computação. Entretanto, parece que os programadores continuaram insensíveis as potencialidades desta nova tecnologia. Talvez isto se deu em parte pelas limitações tecnológicas que se apresentavam na época, mas o mais provável é que eles ainda continuavam escravos dos antigos conceitos de programação, que acreditamos acontece até os dias de hoje. Podemos tomar como exemplo, as ferramentas "gráficas" que se seguiram nesta época e que constituem o fundamento de muitas que usamos ainda hoje. Tínhamos os editores esquemáticos, onde editávamos os diagramas dos circuitos que estavam em desenvolvimento. Quando era necessário efetuar alguma coisa realmente útil como por exemplo uma verificação (simulação), era necessário gerar um *netlist* ("entrada"), aplicá-lo ao simulador ("filtro") para se obter os resultados ("saída"). E analisá-lo, tomando-se estes resultados ("nova entrada") e aplicá-lo a um gerador de formas de onda ("filtro") para que fosse gerado uma representação gráfica ("saída"), conforme ilustra a Figura 1. Ou seja, apesar deste grande recurso os programas nunca deixaram de funcionar desta mesma forma. A forma tradicional da ciência da computação.

Figura 1. Esquema tradicional de operação de programas.

Esta forma de operação, mostrou-se efetiva nos últimos 40 anos mas já tem mostrado sinais de cansaço no decorrer da última década. Este modelo de computação surgiu numa época em que os computadores eram máquinas praticamente inacessíveis a usuários comuns, e que somente especialistas gabaritados podiam operar. Naquela época também o custo de cada equipamento era tão alto que o sua existência somente poderia ser justificada se o seu índice de utilização fosse de 100%, 24 horas por dia.

Sob estas circunstâncias é plenamente justificado o modelo acima, que aliado aos modestos desempenhos existentes, permitiam as operações em *batch* e *background* sem intervenção dos operadores. Entretanto, o que observamos nas últimas duas décadas foi o aparecimento das máquinas de uso pessoal, que possibilitou ao usuário comum o acesso a estes equipamentos. Apesar do tremendo desenvolvimento tecnológico a visão dos programadores continuaram imersas na era dos *mainframes*. Para o usuário comum, os programas continuam difíceis de serem operados, pouco intuitivos, e longe da realidade dos usuários.

Os aperfeiçoamentos que presenciamos atualmente, são modestos se considerássemos as reais potencialidades da tecnologia existente. O único inconveniente seria os interesses econômicos das corporações que dominam o setor. Veremos na seção 5, que apenas uma mudança nos paradigmas de implementação pode abrir uma grande possibilidade de desenvolvimento, sem as inconveniências que estamos apontando até o momento.

O objetivo deste trabalho refere-se principalmente a metodologia de desenvolvimento de sistemas digitais em geral, mas os princípios básicos da metodologia podem ser facilmente estendidos para outras áreas de conhecimento que não estejam diretamente ligadas a ciência da computação mas que carecem de ajuda computacional. A critica principal diz respeito ao modo operacional das ferramentas disponíveis atualmente e da distância conceitual entre programas e usuários, que fazem com que estes últimos participem como meros observadores em seus ramos de atuação.

A metodologia orientada ao projetista (DO), consiste no uso de uma série de recursos para aproximar o universo de aplicação da realidade computacional. Esta aproximação é feita utilizando uma linguagem de programação orientada a objetos (Self [11]), cuja concepção peculiar e avançada permite a implementação quase sem esforço destes recursos. O conceito de programa é diluído num ambiente Self, o que existe é somente um conjunto de objetos que exibe uma dinâmica entre si. Este comportamento é que traduz em operações e processamentos. Self apresenta também um ambiente de desenvolvimento gráfico [2] que reforça o conceito de objetos concretos e acessíveis, permitindo que a programação seja gráfica e textual, dependendo da conveniência do momento. Isto abre uma série de possibilidades com relação ao desenvolvimento de sistemas, os elementos computacionais podem ser encapsulados em termos de elementos comuns ao domínio de especialização do usuário, de forma a facilitar a sua compreensão e uso. Isto pode ser reforçado graficamente, de forma a tornar o desenvolvimento mais agradável e rápido para o usuário. O segundo pilar da metodologia é o conceito de Jogo aplicado ao desenvolvimento, (GLD) "*Game Like Development*". A linguagem Self nos fornece o modelo computacional para a implementação da metodologia através do conceito de objetos concretos, o conceito de Jogo por sua vez determina como devem se apresentar as aplicações em relação ao agente humano, o qual não mais é considerado como elemento separado do processo de desenvolvimento.

4. CONCEITO DE *JOGO* NO DESENVOLVIMENTO

Nesta seção abordaremos uma parte importante da metodologia que é o conceito de *Jogo* aplicado ao desenvolvimento (GLD). Para entender melhor o conceito de *Jogo*, emprestaremos alguns conceitos e observações da filosofia [4] como também invocaremos o senso comum que cada um de nós possuímos em relação ao tema. *Jogo* não é algo estranho para ninguém, de fato ele tem estado presente nas nossas vidas desde a infância, fazendo parte das principais etapas de desenvolvimento do individuo. Inclusive muitos animais apresentam comportamento que indicam a existência de jogos com finalidades diversas. Através dos jogos desenvolvemos qualidades específicas que nos são úteis na vida adulta ou que nos ajudam a contornar situações difíceis possibilitando relaxamento, distração, diversão. Para não entrarmos em infinitas discussões filosóficas, faremos a seguir alguns comentários que achamos pertinentes em relação ao *Jogo* e a contrapartida em relação a metodologia DO.

- Apesar de seu caráter primordialmente lúdico, o *Jogo* possui uma seriedade própria que independe dos elementos que o jogam. Esta seriedade determina um mundo onde se desenrolam as ações e movimentos do *Jogo*. Todos devem respeitar esta formalidade, caso contrário não há *Jogo*. Formalidade é um fator sempre positivo quando relacionado a elementos de computação. Um sistema de desenvolvimento deve oferecer este "mundo". Obviamente, para que seja respeitada a ilusão de *Jogo* este mundo deve ser tão fechado quanto possível e limitar o número de elementos estranhos à dinâmica do desenvolvimento.

- O *Jogo* tem uma natureza própria, independente da consciência daqueles que jogam. O sujeito do *Jogo* não são os jogadores, porém o *Jogo*, através dos que jogam, simplesmente ganha representação. Assim como o *Jogo*, o resultado do desenvolvimento não pode estar submetido aos caprichos dos projetistas. Isto significa que as especificações de um projeto fazem parte das especificações do ambiente do *Jogo* de desenvolvimento, de forma a limitar e direcionar as ações do(s) projetista(s) rumo ao objetivo comum.

- Se considerarmos o significado da palavra *Jogo*, freqüentemente associamos a idéia de conjunto de objetos e movimentos entre eles. Da mesma forma, sistemas digitais podem ser entendidos como objetos que apresentam uma dinâmica muito específica entre si. O *Jogo*, por outro lado, deve representar uma ordem, na qual acontece os movimentos, da mesma forma com que a dinâmica do sistema traduz um comportamento coerente.

- Um dos atrativos lúdicos de um *Jogo* é a sua leveza, ou seja, os movimentos do jogo não devem exigir esforço. Isto convida o jogador a explorar possibilidades e o mantém em foco no jogar. Este é um ponto essencial da metodologia, a inexistência de esforços. Cada movimento, traduzida por uma determinada ação no processo de desenvolvimento, deve aludir a falta de dificuldade. Ou seja, cada ação deve ser para o projetista simples e imediata, permitindo que o mesmo permaneça centrado na tarefa de desenvolvimento. Obviamente, esta leveza não precisa ter um caráter real, apenas uma alusão a falta de esforços, assim como num jogo de monopólio pode-se comprar/construir um conjunto industrial baseados apenas no resultado de um dado.

- Um outro atrativo do *Jogo* é o caráter lúdico da competição, onde o vaivém dos movimentos, livre de esforços, produz situações e condições as vezes além da previsão dos seus jogadores. O *Jogo* surpreende. Um outro caráter importante da metodologia é reagir imediatamente às modificações ou movimentos efetuados pelos projetistas de forma a oferecer o mais rápido possível as informações e conseqüências decorrentes do mesmo.

- O atrativo do *Jogo*, a fascinação que exerce, reside no fato de que o *Jogo* se assenhoreia do jogador. Da mesma forma o ambiente de desenvolvimento deve cativar o projetista de forma a mantê-lo no "jogo", desempenhando as suas funções até que o objetivo seja alcançado.

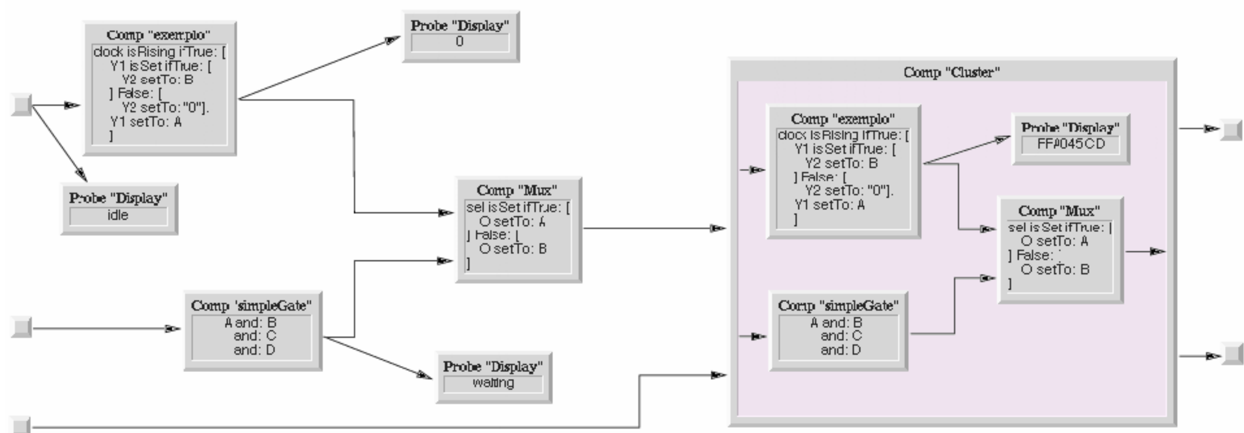

Figura 2 Exemplo de uma descrição de hardware onde representações gráficas e textuais (Self-HDL) se misturam.

- Finalmente, cada jogo coloca uma tarefa ao homem que o joga. Portanto, cabe ao sistema de desenvolvimento colocar este objetivo ao projetista e reforçá-lo em cada momento que for possível, para que o mesmo nunca seja perdido ou disperso.

5. IMPLEMENTAÇÃO DO SISTEMA *DO*

Podemos então delinear algumas características que são fundamentais para a metodologia orientada ao projetista: o sistema deve exibir um nível de integração jamais visto nos *frameworks* convencionais; ele deve ser interativo e interpretado, ou pelo menos interagir com o usuário como se assim fosse; estar voltado ao campo de aplicação; e finalmente, deve lançar mão de todos os recursos disponíveis para manter a atenção do projetista no objeto do seu trabalho. Veremos a seguir como podemos implementar tais características de forma eficiente.

A integração entre ferramentas, ambiente e usuário é um problema bastante complexo. Entretanto, como qualquer outro problema envolvendo complexidade, este também pode ser resolvido adotando-se um modelo de representação adequado. Este modelo é oferecido pela

linguagem Self. O uso de uma linguagem especial para implementação de um *framework* de integração não é uma idéia nova, podemos citar o SKILL da Cadence [1]. Podemos aproveitar as facilidades de desenvolvimento e trabalho com objetos do Self para promover a integração do sistema. Self é uma linguagem de alto nível que possui uma concepção da orientação a objetos muito peculiar que a torna também muito simples. O fato de ser baseada em protótipos e possuir tipos dinâmicos também contribuem muito com essa simplicidade. Todos os elementos em Self são objetos que podem ser acessados, testados, copiados, etc, mesmo durante a execução de um programa. Isto nos remete a segunda característica, a interatividade, num programa em Self um objeto pode ser modificado mesmo quando o mesmo está sendo usado por um programa. É como se um mecânico pudesse mudar as características de uma engrenagem enquanto o motor estivesse em funcionamento. Esta característica é fundamental na implementação da metodologia, pois os objetos podem ser concebidos como implementações reais das primitivas de desenvolvimento, exibindo instantaneamente a sua funcionalidade no instante em que forem instanciados, assim como um componente real é usado numa bancada de teste. Para funcionar como bancada também é preciso que o sistema ofereça um mecanismo de simulação interativa que tenha características mais próximas a emulação do que simulação propriamente dita. Em [7] é mostrado que uma simulação interpretada pode ser quase tão eficiente quanto uma compilada, apesar que o exemplo se referir a simulação lógica podemos facilmente estender o conceito para simulação funcional hierárquica com resultados semelhantes. Finalmente, para que seja criada uma ilusão mais perfeita desta realidade cibernética de desenvolvimento, uma grande quantidade de recursos gráficos devem ser utilizados. Em [2] é apresentada uma implementação gráfica para Self que mantém a programação focalizada nos objetos. Mais uma vez este conceito por ser estendido para o domínio de aplicação de forma a criar o "mundo virtual" de desenvolvimento, sugerido quando falamos do conceito de Jogo.

Como exemplo, vejamos como são implementados alguns objetos fundamentais da metodologia. Num processo de desenvolvimento de *hardware*, é natural e desejável que o projetista pense também em termos de *hardware*. Isto é necessário pois quanto mais próximo do objetivo final do processo mais simples são as ferramentas de auxílio e síntese necessárias. Seguindo este princípio, o objeto[1] principal da metodologia é o objeto "`Comp`", se compararmos com outras HDLs este seria equivalente ao "`Entity`" do VHDL ou "`Module`" do Verilog. Evidentemente na metodologia este objeto tem um escopo muito maior que nestas HDLs. `Comp` pode ter a sua descrição feita em termos funcionais ou estruturais, como pode ser visto na Figura 2. Funcionalmente esta descrição pode ser feita de várias formas diferentes: diagramas de estado, petri-nets, modelos simbólicos e textuais diversos, sendo a Self-HDL a descrição originalmente desenvolvida.

A descrição estrutural consiste numa interconexão de outros objetos `Comp`, com o auxílio dos objetos `Node`. No processo de simulação, estes objetos tem a função de propagação de eventos entre os diversos componentes do sistema. Os `Nodes` herdam o seu modelo de sinal de objetos especiais implementados para tal. Como em Self mesmo a herança pode ser atribuída dinamicamente, os "parents" do `Node` podem ser modificados para refletir o modelo de sinal que for mais conveniente. Atualmente, o sistema está sendo implementado com um modelo simples de quatro valores (0,1,-,X), mas futuramente esperamos que esteja disponível um modelo compatível com o padrão IEEE 1164-1993. Evidentemente, para que existam conexões é necessário que existam `Ports` de entrada e de saída. Todos os `Comps` possuem `Ports` de entrada e de saída, as quantidades variam com a funcionalidade do objeto.

A avaliação do estado de um componente é feita através do envio da mensagem `step` para o mesmo. Independente do tipo de descrição (funcional/estrutural), a mensagem inicia o cálculo das saídas em função das entradas correntes. No caso de uma descrição funcional, isto é feito seguindo diretamente o procedimento especificado pela descrição; no caso estrutural, uma lista de *scheduling* coordenará seletivamente o envio de novas mensagens `step` para os sub-componentes da descrição. A cada saída computada e modificada um novo evento será gerado e acrescentado a lista de *scheduling*, Figura 3. Este esquema é tipicamente *event-driven*, entretanto adotamos otimizações sugeridas em [7] que nos permitem resultados interessantes. Um `Comp` que não seja alterado por um determinado evento não precisa ser incluído na lista de *scheduling*, isto permite por exemplo que simulações *cycle based* possam ser feitas usando o mesmo mecanismo. Customizando os objetos `Comp` e indicando que estes são sensíveis somente a certos `Ports` ("*clocks*"), podemos facilmente transformar uma simulação *event-driven* em *cycle based*, ou mesmo combinar as duas num dado sistema.

---

[1] Faremos todas as descrições em termos de objetos, uma vez que em Self não existe o conceito de classes. Objetos podem herdar de outros objetos (herança múltipla), e são usados/criados a partir de cópias de outros objetos especiais mantidos especialmente para este fim, são os chamados "protótipos".

Até o momento, a implementação não parece muito diferente de outros sistemas. Entretanto como dissemos, a grande diferença encontra-se na integração de ferramentas e na interação com o usuário. Suponha por exemplo, o fluxo ideal de trabalho na metodologia DO e como ele opera com os objetos apresentados. Suponha que estejamos trabalhando num projeto usando uma das descrições de alto nível disponíveis, após as verificações iniciais (simulações) é necessário uma verificação mais efetiva (formal) para que seja possível passar para as próximas etapas de desenvolvimento. Isto é feito simplesmente enviando um mensagem conveniente ao objeto `Comp`. Por exemplo, `checkProp`. Esta mensagem pode consistir de uma verificação de propriedades previamente configuradas no início do projeto. Uma vez confirmada a consistência do modelo, passaríamos a fase de síntese de alto nível, onde baseados na descrição funcional seria gerado um modelo arquitetural (estrutural) do componente. Mais uma vez isto seria resultado de uma simples mensagem, por exemplo: `archGen`. O modelo gerado não iria criar um outro objeto, mas sim acrescentar ao `Comp` original uma nova descrição, desta vez estrutural. `Comp` pode ter quantas descrições se fizerem necessárias, a descrição recém gerada passaria a ser a descrição corrente e a tomada como base seria a descrição anterior. Uma vez exercitada a nova descrição (simulada), uma nova etapa de verificação formal seria necessária, desta vez comparando as duas implementações, isto seria através da mensagem `verify`. Poderíamos seguir sucessivamente estes passos até obter uma descrição do `Comp` original em termos de `Comps` em nível RTL. Neste ponto poderíamos facilmente obter uma descrição HDL convencional através de uma mensagem como `VHDLGen`, ou `VerilogGen`, que se encarregaria de gerar uma descrição HDL que pode ser sintetizada do componente projetado.

Um ponto importante a ser lembrado é que implementando o `framework` em Self, é possível uma funcionalidade exatamente da forma com que foi proposta. Cada uma destas mensagens pode ser simplesmente um botão no ícone de representação de `Comp`, e que só estaria disponível quando e se as condições necessárias ao seu funcionamento estiverem também disponíveis. Não seriam necessários conceitos como arquivos, nomes de arquivos, diretórios, sintaxe de comandos e etc, pois tudo estaria acessível e organizado automaticamente através do ícone de `Comp`.

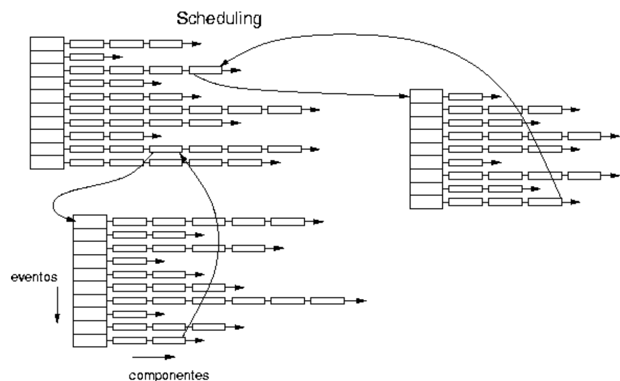

Figura 3 Esquema de scheduling de avaliação de componentes

A interatividade e característica de imersão apontada na seção 4 através do conceito de *Jogo*, é obtida inicialmente através da implementação e manipulação de ícones dos elementos de projeto. Os elementos gráficos têm uma importância muito grande nesta metodologia, não só como elementos de organização como foi apontado anteriormente, mas como elemento essencial para a criação da “realidade virtual” de projeto. Por exemplo, através do mesmo ícone o projetista poderia interagir com `Comp` durante uma simulação. Usando objetos especiais, os `Observers`, que o projetista poderia conectar-se aos `Ports` de um componente e observar o resultado da simulação a medida em que esta se desenrola. Obviamente, um `Observer` poderia ser customizado de forma a formatar o(s) ponto(s) de medição de acordo com as conveniências do projetista, e assim parecer um instrumento de medida, painel de controle e etc. Isto também é muito interessante, pois permite ao projetista formatar e mascarar a implementação de um dado sistema e permitir apresentações de simulações reais para equipes menos ligadas aos detalhes de implementação do *hardware*, como: equipes de desenvolvimento de *software*, *marketing* e negócios.

6. CONCLUSÃO

Apresentamos neste artigo os conceitos principais de uma Metodologia Orientada ao Projetista para o desenvolvimento de sistemas digitais. Vimos que conceitualmente esta metodologia pode ser estendida a outras áreas de conhecimento, porém o objetivo principal deste trabalho é o desenvolvimento de um *framework* para desenvolvimento de sistemas em nível RTL, arquitetural e superior. Foi mostrado como alguns pontos chaves da metodologia estão sendo implementados utilizando a linguagem Self. O trabalho encontra-se nas fases iniciais de desenvolvimento, onde se pretende especificar e implementar um conjunto de objetos de *hardware* que caracterize o nível RTL de descrição, e o mecanismo

básico de inferência para a simulação. Ambas as implementações terão contrapartida textual e gráfica e visam principalmente manter o foco de atenção do projetista nos objetos de *hardware*, sem se preocupar com a interface.

Futuramente planejamos a implementação de um gerador de código HDL padrão (VHDL/Verilog) para propósitos de síntese automática, e também a inclusão de outras representações de auto nível visando talvés *high-level synthesis*.

7. REFERÊNCIAS

[1] CADENCE. SKILL Language Reference Manual. Cadence Design Systems, Inc, June 1989.

[2] CHANG, B.-W., UNGAR, D., AND SMITH, R. B. Visual Object-Oriented Programming. Prentice-Hall, 1995, chapter: Getting Close to Objects: Object-Focused Programming Environments, pp. 185–198.

[3] FLAKE, P. L., DAVIDMANN, S. J., AND KELF, D. J. Measuring design languages for system-on-chip design. Tech. Rep., Co-Design Automation, Inc., San Jose, CA 95113-1295, 2000.

[4] GADAMER, H.-G. Verdade e Método: Traços fundamentais de uma hermenêutica filosófica. Editora Vozes, 1999.

[5] ITRS. International Technology Roadmap for Semiconductors - design – 2001 Edition. http://public.itrs.net/Files/2001ITRS/Home.htm, 2001.

[6] KUHN, T., AND ROSENSTIEL, W. Java based object oriented hardware specification and synthesis. In Proceedings of the ASP-DAC 2000. Asia and South Pacific Design Automation Conference, 2000, pp. 579–581.

[7] MAURER, P. M. Is compiled simulation really faster than interpreted simulation? In Proceedings of the 9th International Conference on VLSI Design, 1996, pp. 303–306.

[8] SHRIVER, B., AND SMITH, B. The Anatomy of a High-Performance Microprocessor: A System Perspective. IEEE Computer Society, 1998.

[9] SYNOPSYS, I. System C - Version 2.0 - User's Guide. http://www.systemc.org, 2002.

[10] THON, L. E., AND BRODERSEN, R. W. C to silicon compilation. In Proceedings of the 1992 IEEE Custom Integrated Circuits Conference (1992), pp. 11.7.1–11.7.4.

[11] UNGAR, D., AND SMITH, R. B. SELF: The power of simplicity. LISP AND SYMBOLIC COMPUTATION: An International Journal. Kluwer Academic Publishers 4, 3 (June 1991).